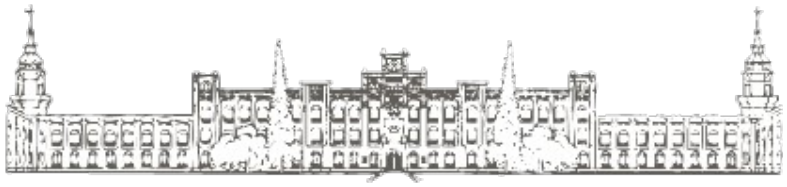

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

КАФЕДРА ІНФОРМАТИКИ ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ

Курсова робота з освітнього компоненту

«Технології паралельних обчислень. Курсова робота»

Тема: **Алгоритм BFS та його паралельна реалізація з використанням мови C++**

Керівник:
проф. Стеценко Інна В’ячеславівна

Виконавець:
Панченко Сергій Віталійович
студент групи ІП-11
залікова книжка № _____

«Допущено до захисту»
_____

«___» _____ 2024 р.

Захищено з оцінкою
_____

Члени комісії:

_____ Інна СТЕЦЕНКО

_____ Олександра ДИФУЧИНА

_____

«23» травня 2024 р.

Київ – 2024

ЗАВДАННЯ

1. Виконати огляд існуючих реалізацій алгоритму BFS, послідовних та паралельних. Зробити висновок про актуальність дослідження.
2. Виконати розробку послідовного алгоритму BFS використанням мови C++. Дослідити швидкодію алгоритму при зростанні складності обчислень та зробити висновки про необхідність паралельної реалізації алгоритму.
3. Виконати розробку паралельного алгоритму BFS використанням мови C++.
4. Виконати тестування алгоритму, що доводить коректність результатів обчислень.
5. Виконати дослідження швидкодії паралельного алгоритму при зростанні кількості даних для обчислень.
6. Виконати експериментальне дослідження прискорення розробленого алгоритму при зростанні кількості даних для обчислень. Реалізація алгоритму вважається успішною, якщо прискорення більше 1.2.
7. Зробити висновки про переваги паралельної реалізації обчислень для алгоритму, що розглядається у курсовій роботі, та програмних засобів, які використовувались.

АНОТАЦІЯ

Пояснювальна записка до курсової роботи: 47 сторінок, 6 рисунків, 3 таблиці.

Об’єкт дослідження: задача паралельного пошуку найкоротшого шляху у графі.

Мета роботи: теоретично дослідити паралельні методи пошуку шляху у графі за допомогою алгоритму BFS; переглянути відомі їхні реалізації; спроектувати, реалізувати, протестувати послідовний та паралельний алгоритми; дослідити ефективність паралелізації програми;

Виконана програмна реалізація паралельного та послідовного алгоритму пошуку шляху у графі, проведено аналіз їх ефективності.

Ключові слова: ПОШУК ШЛЯХУ У ГРАФІ, BFS, КАНАЛИ, ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ.

ЗМІСТ

ВСТУП

У сучасному інформаційному суспільстві важлива роль відводиться оптимізації алгоритмів для вирішення різноманітних завдань, зокрема, задач пошуку шляхів у графі. Алгоритм пошуку в ширину (BFS) визначається як один із найбільш ефективних та широко застосовуваних для вирішення подібних задач. Використання BFS виявляється актуальним у багатьох сферах, таких як штучний інтелект, робототехніка, комп'ютерні ігри та навігація.

З появою багатоядерних процесорів та розподілених систем виникає потреба в розробці ефективних паралельних алгоритмів, спрямованих на прискорення обчислень. Саме у цьому контексті виникає ідея дослідження можливості паралельної реалізації алгоритму BFS за допомогою мови програмування C++.

Об'єктом даної курсової роботи є вивчення та аналіз підходів до паралельної реалізації алгоритму BFS для графів різних розмірностей. Враховуючи особливості BFS, що базується на результатах попередніх ітерацій, основним завданням є розробка ефективної паралельної реалізації. Також в рамках роботи буде розглянуто та проаналізовано ефективність різних методів паралельного виконання алгоритму з урахуванням його особливостей.

Окрім цього, планується розробка та аналіз нового підходу до паралельної реалізації, який сприяє швидкому знаходженню шляхів у графі, особливо при збільшенні розмірності матриці. Отримані результати і висновки будуть важливим внеском у розуміння проблеми паралельного програмування та оптимізації алгоритмів в сучасних умовах розвитку технологій.

1 ОПИС ПОСЛІДОВНОГО АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ

Алгоритм BFS достатньо нескладно розпалалелити, тому окрім класичної реалізації BFS існує паралельна версія BFS зі спільною пам’яттю, тобто чергою, для всіх потоків. У даній роботі будуть розглянуті реалізації послідовна, паралельна зі спільною чергою, а також власні варіції.

1.1 Послідовний BFS

Алгоритм BFS (Breadth-First Search) є ефективним методом для виявлення шляхів у графі та знаходження найкоротших відстаней від стартової вершини до всіх інших вершин. У початковій вершині встановлюється маркер відвіданої, і сусіди цієї вершини додаються до черги. Процес повторюється, доки не буде відвідано всі вершини графа, або знайдено вершину, що відповідає певним критеріям.

1.2 Паралельний BFS зі спільною чергою

У паралельній реалізації алгоритму BFS, кожен потік виконує пошук з різних стартових вершин, використовуючи спільну чергу для координації та обміну даними між потоками. Кожен потік вибирає вершину з черги, додає її сусідів до цієї черги, і позначає їх як відвідані. Цей процес продовжується до тих пір, поки всі вершини графа не будуть відвідані усіма потоками, або знайдено вершину, що відповідає певним критеріям.

2 РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ

У рамках даного розділу проводиться детальний огляд процесу розробки послідовного алгоритму BFS. Визначаються основні кроки та етапи, необхідні для створення функціонального та ефективного алгоритмічного рішення. При цьому здійснюється аналіз особливостей графових структур, що може впливати на вибір оптимальних стратегій алгоритму.

Основний акцент розділу робиться на вивченні та порівнянні часових та просторових характеристик розробленого послідовного алгоритму. Це дозволяє визначити його ефективність та потенційні області оптимізації.

2.1 Проектування послідовного алгоритму

Відповідно до теорії, описаної в пунткі 1.1, було розроблено алгоритм BFS. Він використовує хеш-таблицю для зберігання відвіданих вершин, де ключ — вершина, значення — прапор про відвідування; чергу — для зберігання фронтиру — сукупності сусідніх вершин.

2.2 Реалізація послідовного алгоритму

Для відображення графа достатньо використати std::unordered_map, де ключами будуть вершини, значеннями — вектори з сусідніх вершин. Сама вершина — це певний шаблонний тип, що задовольняє стандарний концепт std::regular (тобто тип має реалізовувати конструктор за замовчуванням, копіювання та оператор порівняння), а також має бути спеціалізована функція std::hash для даного типа. На рисунку 2.1 можна побачити вимоги на типи.

```
template<typename T>
concept CBFSUsable = std::regular<T> and requires(T value) {
        {std::hash<T>{}(value)} -> std::same_as<std::size_t>;
    };
```

Рисунок 2.1 Вимоги до шаблонного типа та графа

2.2.1 TBaseBFSMixin

Будь-який клас алгоритму BFS наслідується від шаблонного класа

TBaseBFSMixin, який побудований за принципом CRTP (Curiously Recurring Template Pattern). CRTP дозволяє використовувати статичний поліморфізм, який є більш типізовано безпечним, та реалізовувати поліморфні функції без використання ключового слова virtual, який накладає додаткову ціну на виклик віртуальної функції. На рисунку 2.2 можна побачити оголошення класа TBaseBFSMixin.

```
template<CBFSUsable T, typename Derived>
class TBaseBFSMixin {
    public:
    template<typename... Args>
    static std::optional<std::vector<T>> Do(const AGraph<T>& graph, const T& start, const T& end, Args&&... args);

    protected:
    TBaseBFSMixin(const AGraph<T>& graph, const T& start, const T& end);

    protected:
    std::optional<std::vector<T>> Execute();

    protected:
    const Derived* self() const;
    Derived* self();

    protected:
    template<typename ValueType>
    std::vector<T> DeterminePath(const std::unordered_map<T, ValueType>& predecessorNodes) const;

    protected:
    const AGraph<T>& m_refGraph;
    const T& m_refStart;
    const T& m_refEnd;
};
```

Рисунок 2.2 Оголошення класа TBaseBFSMixin

TBaseBFSMixin побудовний таким чином, що будь-який дочірній алгоритм надає користувачу лише один статичний метод Do, усе інше - приховане, що робить інтерфейс досить простим.

На рисунку 2.3 TBaseBFSMixin неявно вимагає від дочірніх типів реалізувати метод PredecessorNodesImpl, що має опціонально повертати таблицю відвідування.

```
template<CBFSUsable T, typename Derived>
std::optional<std::vector<T>> TBaseBFSMixin<T, Derived>::Execute() {
    if(m_refStart == m_refEnd) return std::vector{m_refStart, m_refEnd};
    const auto result = self()->PredecessorNodesImpl();
    if(not result) return std::nullopt;
    return DeterminePath(result.value());
}
```

Рисунок 2.3 Накладання вимоги реалізувати метод PredecessorNodesImpl

TBaseBFSMixin зберігає в собі посилання на початкову вершину, кінцеву, а також граф. До того ж має метод DeterminePath, який повертає шлях від початкової до кінцевої вершини. На рисунку 2.4 наведена його реалізація.

```
template<CBFSUsable T, typename Derived>
template<typename ValueType>
std::vector<T> TBaseBFSMixin<T, Derived>::DeterminePath(
    const std::unordered_map<T, ValueType>& predecessorNodes) const {
    auto path = std::vector<T>{this->m_refEnd};
    auto currentNode = path.front();
    while(currentNode != this->m_refStart) {
        currentNode = predecessorNodes.at(currentNode).second;
        path.push_back(currentNode);
    }
    std::reverse(path.begin(), path.end());
    return path;
}
```

Рисунок 2.4 Реалізація метода DeterminePath

2.2.2 TSequentialBFS

Послідовний алгоритм BFS реалізований у вигляді дочірнього класа TBaseBFSMixin TSequentialBFS. На рисунку 2.5 та 2.6 відповідно він визначає метод CreateVisitorMap, що створює хеш-таблицю з ключами-вершинами та значеннями — прапорами відвідування, та реалізовує метод PredecessorNodesImpl.

```
template<CBFSUsable T>
TSequentialBFS<T>::AVisitorMap TSequentialBFS<T>::CreateVisitorMap() const {
    auto visitorMap = std::unordered_map<T, std::pair<bool, T>>();
    visitorMap.reserve(this->m_refGraph.size());
    for(const auto& [key, _] : this->m_refGraph) {
        visitorMap.insert_or_assign(key, std::make_pair(false, T()));
    }
    return visitorMap;
}
```

Рисунок 2.5 Визначення метода CreateVisitorMap

У методі PredecessorNodesImpl спочатку створюється черга, або фронтир вершин, таблиця відвідування; у циклі, доки черга не пуста, або не знайдена кінцева вершина, алгоритм обходить кожну вершину фронтира, позначає її обійденою, дивиться її сусідів, додає їх у чергу, якщо вони не були раніше відвідані, повторює

цю операцію знову.

```
template<CBFSUsable T>
std::optional<typename TSequentialBFS<T>::AVisitorMap>
    TSequentialBFS<T>::PredecessorNodesImpl() const {
    auto queue = std::queue<T>({this->m_refStart});
    auto visitorMap = CreateVisitorMap();
    auto isFoundEndNode = false;
    while(not queue.empty() and not isFoundEndNode) {
        const auto currentNode = std::move(queue.front());
        queue.pop();
        for(const auto& neighbour : this->m_refGraph.at(currentNode)) {
            const auto neighbourIt = visitorMap.find(neighbour);
            if(not neighbourIt->second.first) {
                neighbourIt->second.first = true;
                neighbourIt->second.second = currentNode;
                if(neighbour == this->m_refEnd) {
                    isFoundEndNode = true;
                    break;
                }
                queue.push(neighbour);
            }
        }
    }
    if(not isFoundEndNode) return std::nullopt;
    return visitorMap;
}
```

Рисунок 2.6 Реалізація метода PredecessorNodesImpl

2.3 Тестування послідовного алгоритму

Тестування усіх алгоритмів проводиться за допомогою бібліотеки googletest, яка є стандартом тестування для багатьох C++ проектів. Бібліотека надає класи, методи, макроси тестування, щоб полегшити та зробити більш зрозумілим тестування.

Для початку потрібно визначити метод створення графа. На рисунку 2.7 можна побачити, що граф — це квадрат N*N вершин, де кожна вершина знає про найближчі вершини по горизонталі, вертикалі та діагоналях.

На рисунку 2.8 можна побачити визначення метода Create2DGrid. У циклі з індекса визначається координата вершини, далі до координати вершини додаються

числа -1, 0, 1, потім перевіряється, чи утворена вершина не виходить за межі сітки.

Коректність шляху перевіряється методом IsPathValid, який перевіряє, чи кожна вершина знаходиться в графі, чи кожний наступний елепент шляху є сусідом поточної вершини. На рисунку 2.9 можна побачити визначення даного методу.

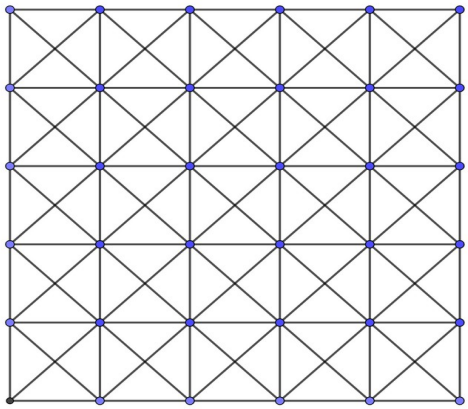

Рисунок 2.7 Вигляд графа для тестування

```
std::unordered_map<unsigned, std::vector<unsigned>>
TTestBFSFixture::Create2DGrid(const unsigned int size) {
    auto grid = std::unordered_map<unsigned, std::vector<unsigned>>();
    const auto totalSize = size * size;
    grid.reserve(totalSize);
    for(auto index = 0u; index < totalSize; ++index) {
        const auto x = static_cast<int>(index % size);
        const auto y = static_cast<int>(index / size);
        const auto utmost = static_cast<int>(size) - 1;
        auto neighbourIndexes = std::vector<unsigned>();
        for(auto deltaY = -1; deltaY <= 1; ++deltaY) {
            const auto newY = y + deltaY;
            if(newY < 0 or newY > utmost) continue;
            const auto base = static_cast<unsigned>(newY) * size;
            for(auto deltaX = -1; deltaX <= 1; ++deltaX) {
                if(deltaY == 0 && deltaX == 0) continue;
                const auto newX = x + deltaX;
                if(newX < 0 or newX > utmost) continue;
                const auto offset = static_cast<unsigned>(newX);
                neighbourIndexes.push_back(base + offset);
            }
        }
        grid.insert_or_assign(index, neighbourIndexes);
    }
    return grid;
}
```

Рисунок 2.8 Визначення метода Create2DGrid

```cpp
bool TTestBFSFixture::IsPathValid(
    const std::vector<unsigned int>& path,
    const bfs::AGraph<unsigned int>& graph) {
    for(const auto& [start, end] : path | std::views::pairwise) {
        const auto it = graph.find(start);
        if(it == graph.end()) {
            return false;
        }
        const auto isContain = std::ranges::contains(it->second, end);
        if(not isContain) {
            return false;
        }
    }
    return true;
}
```

Рисунок 2.9 Визначення метода IsPathValid

На рисунку 2.10 можна побачити, що проводиться пошук шляху від вершини з індексом 0, що знаходиться у верхньому лівому куті графа, до виршини lastIndex, що знаходиться у нижньому правому куті графа. Далі вимірюється час у мілсекундах та перевіряється, чи справді створений алгоритмом шлях валідний. Усі результати вимірювання записуються у JSON-форматі, де вказується назва алгоритму, розмір графа, кількість мілісекунд виконання, щоб надалі з допомогою Python можна було побудувати графіки порівняння алгоритмів.

```cpp
constexpr auto totalRepeats = 5u;
const auto sizes = std::vector<unsigned>{2500, 2625, 2750, 2875, 3000, 3125, 3250, 3500, 3635, 3750, 3875, 4000};
//const auto sizes = std::vector<unsigned>{100, 200};
const auto threadsNums = std::vector<unsigned>{2, 3, 4, 5, 6, 7, 8, 9};
for(const auto size : sizes) {
    const auto grid = Create2DGrid(size);
    const auto lastIndex = GetLastIndex(size);
    for(auto i = 0u; i < totalRepeats; ++i) {
        const auto sequentialMillis = static_cast<double>([&grid, &lastIndex, &size]() {
            const auto start = std::chrono::system_clock::now();
            const auto result = bfs::TSequentialBFS<unsigned>::Do(grid, 0, lastIndex);
            const auto delay = std::chrono::system_clock::now() - start;
            const auto millis = std::chrono::duration_cast<std::chrono::milliseconds>(delay).count();
            EXPECT_TRUE(IsPathValid(result.value(), grid));
            WriteToReport(std::format("{{ \"name\": {}, \"size\": {}, \"milliseconds\": {} }}", "Sequential", size, millis));
            return millis;
        }());
```

Рисунок 2.10 Тестування послідовного алгоритму

У таблиці 2.1 наведено залежність часу виконання від розміру сітки. Для кращої репрезентації результатів побудуємо графік, який наведений на рисунку 2.11. Як бачимо, зі збільшенням розмірності графа час також зростає. Проведення тестування на матрицях більшої розмірності не вдається, оскільки їхнє створення

займає багато часу, а також є обмеження в оперативній пам’яті.

Таблиця 2.1 Тестування послідовного алгоритму

Розмір	Час виконання у мілісекундах
2500	5967.6
2625	6643.2
2750	7336.6
2875	9370.8
3000	8698.6
3125	9284.8
3250	10215.0
3500	12120.2
3635	13128.2
3750	14052.8
3875	15112.6
4000	16670.8

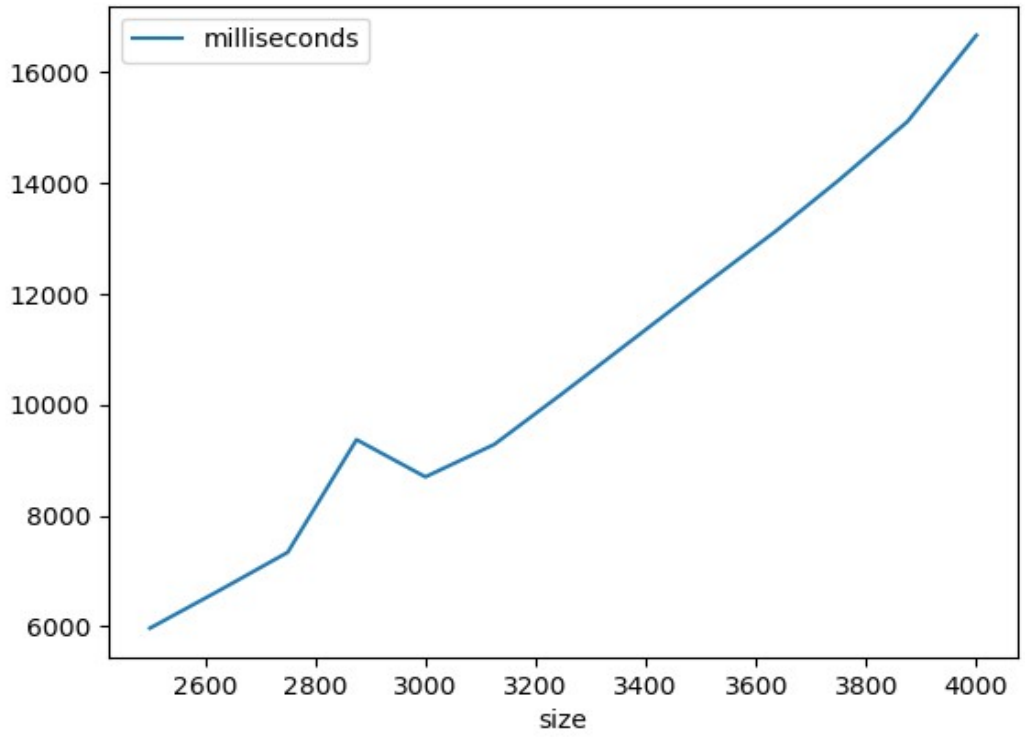

Рисунок 2.11 Графік залежності часу від розміру графа для послідовного алгоритму

Бачимо, що різниця у часі для графів з розмірами 2500 та 4000 відрізняється ледь не втричі, тому є сенс розпалалелити алгоритм.

2.4 Висновок

У даному розділі було проведено детально розробку алгоритму BFS на основі класу TSequentialBFS. Також описали процеси тестування: створення графа, вимірювання часу виконання алгоритму від розміру графів. Результати виконання записуються у JSON-форматі для зручної обробки результатів.

Як побачили час виконання помітно зростає зі збільшенням розміру графа, тому потенціал до покращення з допомогою паралелезації існує.

3 ВИБІР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗРОБКИ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ ТА ЙОГО КОРОТКИЙ ОПИС

Для реалізації паралельного алгоритму BFS я обрав C++. Вона є потужною системною мовою програмування, яка надає високі абстракції над низькорівневими речами без втрати швидкості.

C++ включає в стандартну бібліотеку засоби паралельного програмування, як-от: потоки, м’ютекси, атомарні типи тощо. Для цього він має відповідні файли <thread>, <mutex>, <atomic> тощо.

Також варто розглянути засоби, які існують, але не будуть використані у даній роботі.

Boost.MPI — це файл бібліотеки Boost, що надає ООП обгорти для MPI API мови C. Недоліком даного засобу є те, що він вимагає нестандартний компілятор MPI, який не підтримує нові версії мови C++.

Boost.Asio — це файл бібліотеки Boost, що відповідає за асинхронний вивід та ввід даних у мережі. Цю бібліотеку можна було б використатити для передавання повідомлень між потоками, але, на жаль, Boost.Asio надає можливість створювати канали між потоками тільки у вигляді сокетів та передавати дані тільки як байти без прив’язки до типів.

До того ж сама бібліотека Boost величезна і суттєво збільшить час компіляції проєкту.

4 РОЗРОБКА ПАРАЛЕЛЬНОГО АЛГОРИТМУ З ВИКОРИСТАННЯМ ОБРАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ: ПРОЕКТУВАННЯ, РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ

У рамках даного розділу проводиться детальний огляд процесу розробки паралельних алгоритмів BFS. Як і в розділі 2 визначаються основні кроки та етапи, необхідні для створення функціонального та ефективного алгоритмічного рішення. Детально розглядаються можливість розпаралелювання, вибір структур даних та оптимізаційні підходи, що можуть покращити продуктивність алгоритму. Також буде проведений аналіз результатів тестування та визначені потенційні області оптимізації для забезпечення оптимальної швидкодії паралельного алгоритму BFS.

4.1 Варіанти паралельної реалізації

4.1.1 BFS з спільною чергою

Найпростішим варіатом розпалалелити алгоритм є зробити потокостійкі класи таблиці відвідування та фронтиру та розділити їх між всіма потоками. Однак у такої реалізації є суттєві проблеми.

Розглянемо для початку розділення таблиці відвідування між всіма потоками. Логічно утворити окремий клас TThreadSafeMap, у якому буде м’ютекс та таблиця відвідування. Таким чином буде забезпечено, що тільки один потік буде модифікувати таблицю в певний момент. На рисунку 4.1 зображено схему даної реалізації.

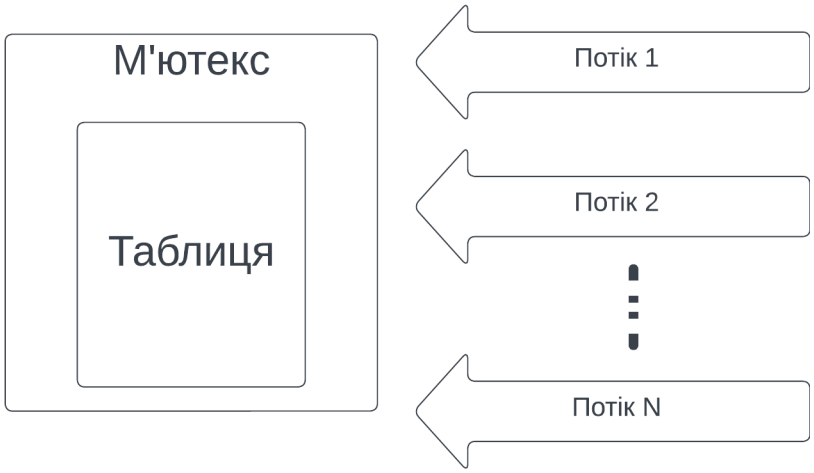

Рисунок 4.1 Таблиця відвідування покрита суцільним м’ютексом

Однак можна зрозуміти, що таблиця буде створена один раз на початку алгориму, і її розмір не буде змінюватися, лише її елементи будуть приймати інші значення. Поглянемо на популярні бази даних, як-от: PostgreSQL, MS SQL, MySQL — усі вони вирішують дану проблему надаючи м’ютекс лише на певну частину даних. Тому набагато ефективніше буде мати в кожному записі атомарний прапор, який буде позначати, чи була вершина раніше відвідана чи ні. На рисунку 4.2 можна побачити схему даної реалізації.

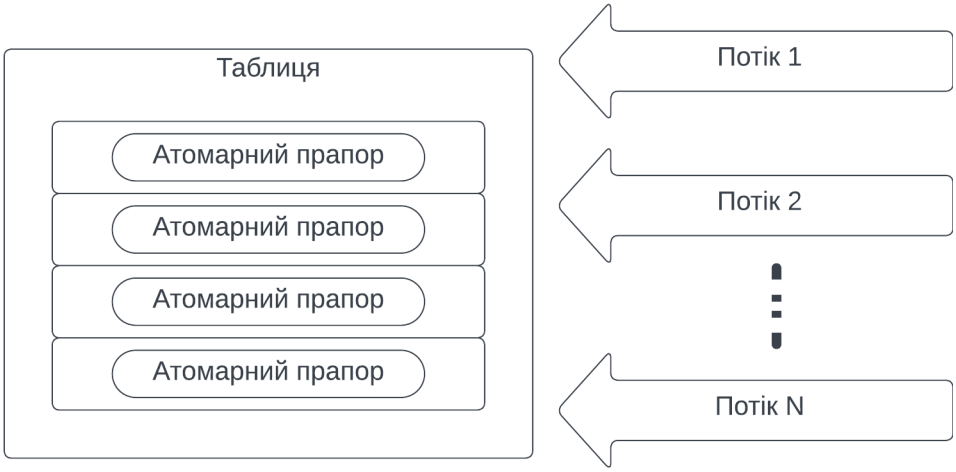

Рисунок 4.2 Таблиця відвідування з атомарними прапорами

Щодо черги, або фронтира, так само не вдається зробити, тому вона буде працювати за першим варіантом, однак для білшої оптимізації буде викрастано не std::mutex зі стандартної бібліотеки, а власно написаний на атомарних прапорах. Це дасть черзі швидше працювати та надавати доступ до модифікації даних іншим потокам.

4.1.2 BFS з повідомленнями

Після багатьох спроб пришвидшити паралельний варіант BFS зі спільною чергою я не був задовлений результатом. Алгоритм і справді виконувався швидше, але коли кількість потоків зростала до 6 і більше, вони часто синхронізувалися і переривали один одного при модифікації фронтиру, тому час виконнання був суттєво більшим ніж за тих самих параметрів у послідовної версії.

Тоді було вирішено піти іншим шляхом: нехай основний потік буде через певний канал спілкуватися з дочірнім потоком те передавати повідомлення про виконання алгоритму. Таким чином буде один раз створено дочірні потоки, їм

будуть надані канали спілкування з основиним потоком; кожен дочірній потік буде обробляти лише свою частину черги, заповнювати власну локальну чергу, передавати її назад основному потоку; потім основний потік збирає докупи результати виконання від усіх потоків, далі надсилає інформацію про те, яку саму частку спільної черги обробити; далі цей алгоритм продовжується, допоки не будуть оброблені всі вершини, або знайдена кінцева.

На рисунку 4.3 показана загальна ідея алгоритму.

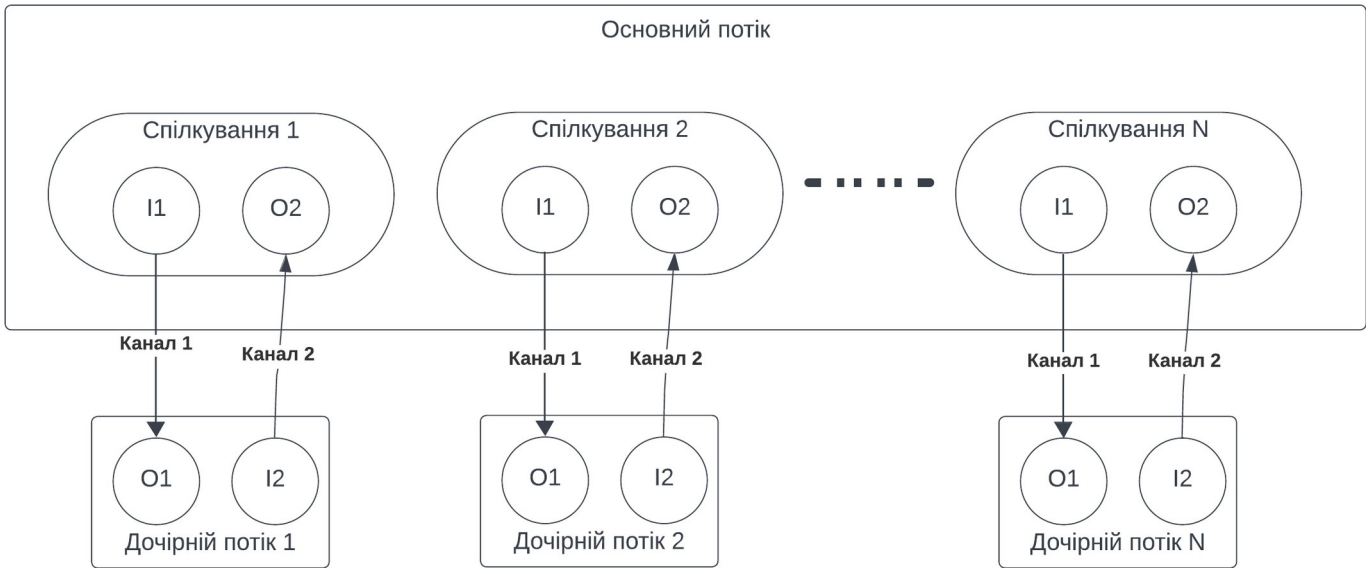

Рисунок 4.3 Загальна ідея BFS з повідомленнями

Отже, написання поточної версії алгоритму складається з декількох пунктів:

— написати певну абстракцію, яку назвемо “канал”, через яку будуть передаватися певні типи;
— написати певний контейнер, який буде зберігати сукупність векторів, тобто результи дочірніх потоків; даний контейнер має ззовні здаватися неперервним шматком даних, до яких можна дістатися за допомогою ітераторів;
— написати загальну схему алгоритму в псевдокоді.

4.2 Проектування та реалізація паралельних алгоритмів

У даному підрозділі будуть розглянуті реалізації класів, що використовуються в паралельних версіях алгоритмів. За потреби будуть зображені схеми для кращого розуміння.

4.2.1 TParallelBFSMixin

Для реалізації описаних алгоритмів існує клас TParallelBFSMixin, який додає поле кількості потоків в алгоритмі та відповідає за створення атомарної таблиці відвідування. На рисунку 4.4 можна побачити даний клас. Атомарна таблиця — це std::unordered_map, де ключами є вершини, значеннями — пара атомарного прапора та наступної у шляху сусідньої вершини.

```
template<CBFSUsable T, typename Derived>
class TParallelBFSMixin : public TBaseBFSMixin<T, Derived> {
    protected:
    TParallelBFSMixin(const AGraph<T>& graph, const T& start,
        const T& end, const unsigned threadsNum);

    protected:
    using AVisitorMap = std::unordered_map<T, std::pair<std::atomic_flag, T>>;

    protected:
    AVisitorMap CreateVisitorMap() const;

    protected:
    const unsigned m_uThreadsNum = 0;
};

template<CBFSUsable T, typename Derived>
std::unordered_map<T, std::pair<std::atomic_flag, T>>
    TParallelBFSMixin<T, Derived>::CreateVisitorMap() const {
    auto visitorMap = std::unordered_map<T, std::pair<std::atomic_flag, T>>();
    visitorMap.reserve(this->m_refGraph.size());
    for(const auto& [key, _] : this->m_refGraph) {
        visitorMap.emplace(std::piecewise_construct,
            std::forward_as_tuple(key), std::forward_as_tuple());
    }
    return visitorMap;
}

template<CBFSUsable T, typename Derived>
TParallelBFSMixin<T, Derived>::TParallelBFSMixin(const AGraph<T>& graph, const T& start,
    const T& end, const unsigned threadsNum)
    : m_uThreadsNum{std::min(threadsNum, std::jthread::hardware_concurrency())},
    TBaseBFSMixin<T, Derived>(graph, start, end) {}

}
```

Рисунок 4.4 Реалізація класу TParallelBFSMixin

4.2.2 TAtomicQueue

TAtomicQueue — це потокостійка черга, особливістю якої є те, що використовує для блокування доступу до даних не м’ютекси, а атомарні прапори, оскільки вони зміна їхніх значень відбувається швидше ніж замикання чи відімкнення м’ютекса. На рисунку 4.5 зображена реалізація класу.

```
template<typename T>
class TAtomicQueue {
    public:
    TAtomicQueue()=default;

    public:
    template<typename U>
    void Push(U&& value);
    std::optional<T> Pop();

    protected:
    std::atomic_flag m_xFlag;
    std::queue<T> m_qQueue;
};

template<typename T>
template<typename U>
void TAtomicQueue<T>::Push(U&& value) {
    while(m_xFlag.test_and_set());
    m_qQueue.push(std::forward<U>(value));
    m_xFlag.clear();
}

template<typename T>
std::optional<T> TAtomicQueue<T>::Pop() {
    while(m_xFlag.test_and_set());
    if(m_qQueue.empty()) {
        m_xFlag.clear();
        return std::nullopt;
    }
    auto popped = std::move(m_qQueue.front());
    m_qQueue.pop();
    m_xFlag.clear();
    return popped;
}
```

Рисунок 4.5 Реалізація класу TAtomicQueue

Як бачимо на прикладах методів Push та Pop, потік очікує та перевіряє

значення прапора доти, доки прапорець знову не очиститься.

4.2.3 TSharedBFS

TSharedBFS — це паралельна версія BFS зі спільною чергою для всіх потоків. Розглянемо реалізацію метода PredecessorNodesImpl на рисунку 4.6.

```
template<CBFSUsable T>
std::optional<typename TSharedBFS<T>::AVisitorMap> TSharedBFS<T>::PredecessorNodesImpl() const {
    auto queue = TAtomicQueue<T>();
    queue.Push(this->m_refStart);
    auto visitorMap = this->CreateVisitorMap();
    auto& isEndNodeFound = visitorMap.find(this->m_refEnd)->second.first;
    auto totalEnqueuedNum = std::atomic_size_t{0};
    {
        auto threads = std::vector<std::jthread>();
        threads.reserve(this->m_uThreadsNum);
        for(auto i = 0u; i < this->m_uThreadsNum; ++i) {
            threads.emplace_back([this, &queue, &visitorMap, &totalEnqueuedNum, &isEndNodeFound]() {
                while(not isEndNodeFound.test()) {
                    if(totalEnqueuedNum.load() >= this->m_refGraph.size()) return;
                    const auto currentNodeOpt = queue.Pop();
                    if(not currentNodeOpt) continue;
                    const auto& currentNode = currentNodeOpt.value();
                    for(const auto& neighbour : this->m_refGraph.at(currentNode)) {
                        const auto neighbourIt = visitorMap.find(neighbour);
                        if(neighbourIt->second.first.test_and_set()) continue;
                        neighbourIt->second.second = currentNode;
                        if(neighbour == this->m_refEnd) return;
                        totalEnqueuedNum.fetch_add(1);
                        queue.Push(neighbour);
                    }
                }
            });
        }
    }
    if(not isEndNodeFound.test()) return std::nullopt;
    return visitorMap;
}
```

Рисунок 4.6 Визанчення методу PredecessorNodesImpl класу TSharedBFS

Перше, що треба обов’язково зазначити, що для потоків використовується клас з C++20 std::jthread(Join-Thread), який автоматично викликає метод join у деструкторі, що надзвичайно зручно.

На початку алгоритму створюється таблиця відвідування, черга з початковою вершиною, лічильник кількості обійдених вершин. Посилання на них надається дочірнім потокам. У всьому іншому алгоритм аналогічний до TSequentialBFS.

4.2.4 TPipeReader, TPipeWriter, TPipeChannel

TPipeChannel — класи, що відповідають за реалізацію каналу передачі даних між потоками. Він складається з входу — TPipeWriter — та виходу — TPipeReader. Обидва кінця каналу володіють посиланнями на дані, що передаються. Самі дані захищені атомарним прапором. Обидва кінці мають методи Write та Read, які очікують виконання доки дані не будуть звільненні на вписування або на зчитування. Для кращого розуміння роботи зображено рисунок 4.7.

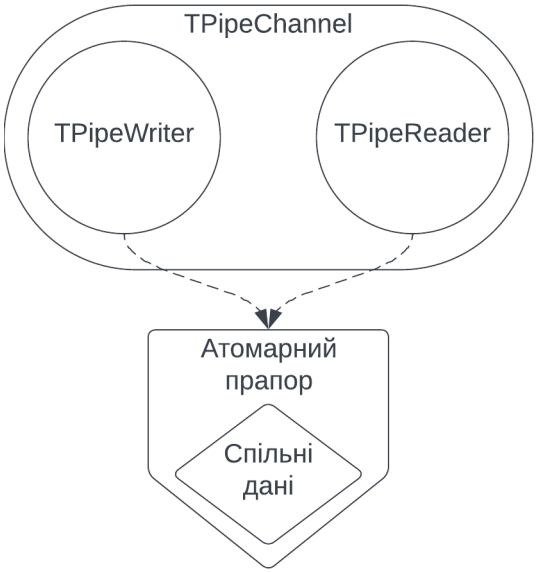

Рисунок 4.7 Схема роботи каналу

Розглянемо C++ реалізацію даної схеми. TPipeChannel — це клас, що зберігає в собі обидва кінця канала, у коструторі він надає каналам спільне посилання на пару з атомарного прапора та даних. Дані мають реалізовувати конструктори за замовчуванням і переміщення. На рисунку 4.8 позначені обмеження. На рисунку 4.9 показується реалізація класа TPipeChannel.

```
template<typename T>
concept CPipeUsable = std::default_initializable<T> and std::movable<T>;
```

Рисунок 4.8 Обмеження на тип в каналах

TPipeWriter та TPipeReader мають лише по одному публічному методу Write та Read і конструктор переміщення, інші конструктори, методи, поля класів інкапсульовані. Обидва класи знають, що їхнім дружнім типом є TPipeChannel, і тільки він може створювати об’єкти даних класів. Це зроблено для того, щоб не можна було окремо створити чи скопіювати або TPipeWriter або TPipeReader, вони

існують лише в парі. Загалом класи однакові за будовою, тому буде показано лише оголошення класу TPipeReader на рисунку 4.10.

```
template<CPipeUsable T>
class TPipeChannel {
    public:
    TPipeChannel();

    public:
    TPipeWriter<T> Writer;
    TPipeReader<T> Reader;
};

template<CPipeUsable T>
TPipeChannel<T>::TPipeChannel() {
    auto data = std::make_shared<std::pair<T, std::atomic_flag>>();
    Writer = TPipeWriter(data);
    Reader = TPipeReader(std::move(data));
}
```

Рисунок 4.9 Визначення класу TPipeChannel

```
template<CPipeUsable T>
class TPipeReader {
    friend class TPipeChannel<T>;

    public:
    ~TPipeReader()=default;
    TPipeReader(TPipeReader&& other) noexcept;
    TPipeReader& operator=(TPipeReader&& other) noexcept;

    public:
    T Read() const;

    protected:
    TPipeReader() = default;
    TPipeReader(const std::shared_ptr<std::pair<T, std::atomic_flag>>& data);
    TPipeReader(const TPipeReader&) = delete;
    TPipeReader& operator=(const TPipeReader&) = delete;

    protected:
    std::shared_ptr<std::pair<T, std::atomic_flag>> m_pData = nullptr;
};
```

Рисунок 4.10 Оголошення класу TPipeReader

4.2.5 TDeque

TDeque — клас, що виконує роль представлення для вектора векторів. Особливістю цього класу є те, що він дозволяє користувачу ітеруватися по ньому, як неперервному контейнеру. Для цього клас має публічний метод Loop, який приймає індекси початку та кінця, а також лямбду, до якої будуть передаватися елементи під час ітерації. На рисунку 4.11 зображений метод Loop. Як бачимо, беруться ітератори внутрішніх векторів, потім як тільки досягається кінець внутрішнього ітератора, зовнішній йде до наступного вектора.

```
template<typename T>
void TDeque<T>::Loop(const size_t begin, const size_t end,
    const std::function<void(const T&)>& func) const {
    if(begin == end) return;
    auto vectorIt = m_vData.begin();
    auto elIt = m_vData.begin()->begin();

    auto delay = begin;
    auto dist = vectorIt->end() - elIt;
    while(delay >= dist) {
        delay -= dist;
        ++vectorIt;
        elIt = vectorIt->begin();
        dist = vectorIt->end() - elIt;
    }

    elIt += delay;

    for(auto i = begin; i < end; ++i, ++elIt) {
        if(elIt == vectorIt->end()) {
            ++vectorIt;
            elIt = vectorIt->begin();
        }
        func(*elIt);
    }
}
```

Рисунок 4.11 Визначення методу Loop

4.2.6 TCommunicationBFS

TCommunicationBFS — це паралельна реалізація BFS з повідомленнями. Розглянемо структуру даного класу.

Для початку розглянемо повідомлення, з допомогою яких потоки будуть спілкуватися між собою. На жаль, через специфіку мови C++ класи не можуть мати внутрішнього простору імен, а також enum’и не можуть в себе приймати структури даних, як в мові Rust. Тому повідомлення відображені у вигляді структур, що містяться всередині структури NMessage. На рисунку 4.12 можна побачити реалізацію.

```
protected:
struct NMessage {
    struct SEndNodeFound {};
    struct SAllNodesEnqueued {};
    struct SQueueView {
        const TDeque<T>* Deque;
        size_t Begin;
        size_t End;
    };
    struct SFrontier {
        std::vector<T> Data;
    };
};
```

Рисунок 4.12 Типи повідомлень

Як бачимо, повідомлення включають:

— SEndNodeFound — знайдена кінцева вершина;

— SAllNodesEnqueued — усі вершини обійдені;

— SQueueView — певна частина фронтиру вершин, які має обійти дочірній потік;

— SFrontier — частина загального фронтиру, утворена дочірнім потоком.

Повідомлення об’єднані у два варіативних типи на рисунку 4.13:

— AParentMessage — повідомлення від основного потоку до дочірніх;

— AChildrenMessage — повідомлення від дочірніх потоків до основного.

```
using AParentMessage = std::variant<
    typename NMessage::SEndNodeFound,
    typename NMessage::SAllNodesEnqueued,
    typename NMessage::SQueueView>;

using AChildrenMessage = std::variant<
    typename NMessage::SEndNodeFound,
    typename NMessage::SFrontier>;
```

Рисунок 4.13 Варіативні типи повідомлень

TCommunicationTask — це клас, що відображає роботу дочірнього потоку та має перевантажений оператор виклику, що робить об’єкт даного класу функтором. На рисунку 4.14 зображене оголешення класу.

```
protected:
class TCommunicationTask {
    public:
    TCommunicationTask(
        const AGraph<T>& graph,
        const T& end,
        TPipeWriter<AChildrenMessage>&& sender,
        TPipeReader<AParentMessage>&& listener,
        typename TCommunicationBFS<T>::AVisitorMap& visitorMap);

    void operator()();

    protected:
    const AGraph<T>& m_refGraph;
    const T& m_refEnd;
    TPipeWriter<AChildrenMessage> m_xSender;
    TPipeReader<AParentMessage> m_xListener;
    typename TCommunicationBFS<T>::AVisitorMap& m_refVisitorMap;
};
```

Рисунок 4.14 Оголошення класу TCommunicationTask

На рисунку 4.15 зображено визначення опереатору виклику TCommunicationTask.

Як бачимо, до кожного дочірнього потоку передається вихід першого каналу та вхід другого каналу для двостороннього спілкування. У циклі на початку кожної ітерації дочірній потік очікує повідомлення, якщо повідомлення не містить SQueueView, тобто частину фронтиру, то це означає, що або всі вершини обійшли, або знайдено кінцеву. Далі при обході своєї частини фронтиру, беруться сусіди

поточної вершини та відбераються до результуючого фронтиру тільки та, які не були відвідані на даний момент. Якщо вершина не відвідана, то її атомарний прапор позначається, та перевіряється, чи вона не є кінцевою. Якщо вершина кінцева, то основному потоку надсилається повідомлення, що треба завершити пошук, інакше надсилається результуючий фронтир.

```
template<CBFSUsable T>
void TCommunicationBFS<T>::TCommunicationTask::operator()() {
    while(true) {
        auto parentMessage = m_xListener.Read();
        if(not std::holds_alternative<typename NMessage::SQueueView>(parentMessage)) return;
        auto frontier = typename NMessage::SFrontier();
        const auto [deque, begin, end] = std::get<typename NMessage::SQueueView>(parentMessage);
        auto isEndNodeFound = false;
        deque->Loop(begin, end, [this, &frontier, &isEndNodeFound](const T& node) {
            for(const auto& neighbour : this->m_refGraph.at(node)) {
                const auto neighbourIt = m_refVisitorMap.find(neighbour);
                if(neighbourIt->second.first.test_and_set())
                    continue;
                neighbourIt->second.second = node;
                if(neighbour == this->m_refEnd) {
                    m_xSender.Write(typename NMessage::SEndNodeFound{});
                    isEndNodeFound = true;
                    return;
                }
                frontier.Data.push_back(neighbour);
            }
        });
        if(isEndNodeFound) return;
        m_xSender.Write(std::move(frontier));
    }
}
```

Рисунок 4.15 Визначення опереатору виклику TCommunicationTask

На рисунку 4.16 зображене оголошення класу TCommunicationCenter, що відповідає за координацію дій дочірніх потоків. Даний клас володіє векторами входів та виходів каналів потоків для двостороннього спілкування. Також у полі класу має загальний фронтир, який буде розподілений між всіма дочірніми потоками; а також розмір графа та лічільник кількості обійдених вершин.

ACommunicationResult — це варіативний тип, який вказує на результат знаходження шляху: або всі вершині обійдені, і нічого не знайдено, або існує шлях до кінцевої вершини.

```
protected:
class TCommunicationCenter {
    public:
    TCommunicationCenter(const size_t graphSize, const T& start,
        std::vector<TPipeWriter<AParentMessage>>&& senders,
        std::vector<TPipeReader<AChildrenMessage>>&& listeners);

    public:
    using ACommunicationResult = std::variant<
        typename NMessage::SEndNodeFound,
        typename NMessage::SAllNodesEnqueued>;

    ACommunicationResult Communicate();

    protected:
    void SendTasks();

    template<typename MessageType>
    MessageType SendMessageToAll();

    protected:
    const size_t m_uGraphSize;
    TDeque<T> m_vDeque;
    std::vector<TPipeWriter<AParentMessage>> m_vSenders;
    std::vector<TPipeReader<AChildrenMessage>> m_vListeners;
    size_t m_uTotalEnqueuedNum = 0;
};
```

Рисунок 4.16 Оголошення класу TCommunicationCenter

Розглянемо публічний метод Communicate на рисунку 4.17. У циклі усім дочірнім потокам надсилаються завдання, потім очікується відповідь від них, якщо знайдено кінцеву ноду, то це повідомлення надсилається дочірнім потокам і вони закінчують своє виконання. Якщо був отриманий частковий фронтир, його розмір додається до загального лічильника, а далі він переміщується у загальний фронтир. Якщо всі вершини обійдені, то це повідомлення надсилається іншим потокам, і вони завершують свою роботу.

На рисунку 4.18 зображене визначення методу SendMessageToAll, який надсилає певний простий тип повідомлення усім дочірнім потокам.

```
template<CBFSUsable T>
TCommunicationBFS<T>::TCommunicationCenter::ACommunicationResult
TCommunicationBFS<T>::TCommunicationCenter::Communicate() {
    while(true) {
        SendTasks();
        auto newDeque = TDeque<T>();
        for(auto& l : m_vListeners) {
            auto message = l.Read();
            switch(message.index()) {
                case VariantIndex<AChildrenMessage, typename NMessage::SEndNodeFound>(): {
                    return SendMessageToAll<typename NMessage::SEndNodeFound>();
                }
                case VariantIndex<AChildrenMessage, typename NMessage::SFrontier>(): {
                    auto frontier = std::get<typename NMessage::SFrontier>(message);
                    m_uTotalEnqueuedNum += frontier.Data.size();
                    if(not frontier.Data.empty()) {
                        newDeque.Push(std::move(frontier.Data));
                    }
                }
            }
        }
        m_vDeque = std::move(newDeque);
        if(m_uTotalEnqueuedNum >= m_uGraphSize) {
            return SendMessageToAll<typename NMessage::SAllNodesEnqueued>();
        }
    }
    return typename NMessage::SAllNodesEnqueued{};
}
```

Рисунок 4.17 Визначення методу Communicate

```
template<CBFSUsable T>
template<typename MessageType>
MessageType TCommunicationBFS<T>::TCommunicationCenter::SendMessageToAll() {
    auto message = MessageType{};
    for(auto& s : m_vSenders) {
        s.Write(message);
    }
    return message;
}
```

Рисунок 4.18 Визначення методу SendMessagesToAll

На рисунку 4.19 зображено визначення методу SendTasks, де між дочірніми потоками рівномірно розподіляється загальний фронтир. Якщо так сталося, що розмір фронтиру меший за кількість потоків, то потокам, яким не вистачило вершин,

надсилаються кінцеві ідекси.

```
template<CBFSUsable T>
void TCommunicationBFS<T>::TCommunicationCenter::SendTasks() {
    const auto dequeSize = m_vDeque.Size();
    const auto threadsNum = m_vSenders.size();
    const auto end = m_vDeque.Size();
    auto begin = size_t(0);
    auto i = std::size_t(0);
    const auto step = std::max(size_t(1), dequeSize / threadsNum);
    const auto stepsNum = std::min(dequeSize, threadsNum - 1);
    for(;i < stepsNum; ++i) {
        const auto next = begin + step;
        m_vSenders[i].Write(typename NMessage::SQueueView{&m_vDeque, begin, next});
        begin = next;
    }
    if(stepsNum == dequeSize) {
        for(;i < threadsNum; ++i) {
            m_vSenders[i].Write(typename NMessage::SQueueView{&m_vDeque, end, end});
        }
    } else {
        m_vSenders[i].Write(typename NMessage::SQueueView{&m_vDeque, begin, end});
    }
}
```

Рисунок 4.19 Визначення методу SendTasks

І кінці розглянемо на рисунку 4.20 визначення методу PredecessorNodesImpl. На початку створюється матриця відвідування, далі між основним потоком та дочірнім створюється два канали для двосторонньої комунікації. Входи першого та виходу другого каналів збираються у вектори. Далі створюється центр комунікації, котрому передаються кінці цих каналів. Викликається метод Communicate, якщо він повернув повідомлення про те, що знайдена кінцева вершина, то повертається таблиця відвідування; якщо повернулося повідомлення про те, що всі вершини обійдені, то повертається пусте значення.

```cpp
template<CBFSUsable T>
std::optional<typename TCommunicationBFS<T>::AVisitorMap>
    TCommunicationBFS<T>::PredecessorNodesImpl() const {
    auto visitorMap = this->CreateVisitorMap();
    auto threads = std::vector<std::jthread>();
    auto senders = std::vector<TPipeWriter<AParentMessage>>();
    auto listeners = std::vector<TPipeReader<AChildrenMessage>>();
    for(auto i = 0u; i < this->m_uThreadsNum; ++i) {
        auto [parentSender, parentListener] = TPipeChannel<AParentMessage>();
        auto [childrenSender, childrenListener] = TPipeChannel<AChildrenMessage>();
        senders.push_back(std::move(parentSender));
        listeners.push_back(std::move(childrenListener));
        threads.emplace_back(TCommunicationTask(this->m_refGraph, this->m_refEnd,
            std::move(childrenSender), std::move(parentListener), visitorMap));
    }
    auto communicationCenter = TCommunicationCenter(
        this->m_refGraph.size(), this->m_refStart,
        std::move(senders), std::move(listeners));
    const auto communicationResult = communicationCenter.Communicate();
    if(std::holds_alternative<typename NMessage::SEndNodeFound>(communicationResult)) {
        return visitorMap;
    }
    return std::nullopt;
}
```

Рисунок 4.20 Визначення методу PredecessorNodesImpl

4.3 Тестування паралельних алгоритмів

У даному підрозділі буде проведено тестування не тільки паралельних алгоритмів, а й структур даних, що допомогли їх реалізувати.

Проведемо за таблицею 4.1 тестування TPipeChannel, TPipeWriter, TPipeReader. На рисунку 4.21 та 4.22 зображено код тесту та результат відповідно.

Таблиця 4.1 Тестування TPipeChannel, TPipeWriter, TPipeReader

Тест	Перевірка роботи TPipeChannel, TPipeWriter, TPipeReader
Номер тесту	1
Початковий стан	Маємо вхід і вихід каналу
Вхідні дані	Число 10
Опис проведення тесту	Створимо два потоки, змусимо перший потік зупинитися на 4 секунди.
Очікуваний результат	Другий буде очікувати отримання даних, тобто числа 10.
Фактичний результат	Число 10 вивелося на екрані.

```
TEST(Pipes, Transfer) {
    auto [w, r] = bfs::TPipeChannel<int>();

    auto sender = std::jthread([ww=std::move(w)]() {
        using namespace std::chrono_literals;
        std::this_thread::sleep_for(4s);
        ww.Write(10);
    });

    auto listener = std::jthread([rr=std::move(r)]() {
        EXPECT_EQ(rr.Read(), 10);
    });
}
```

Рисунок 4.21 Тест працездатності TPipeChannel, TPipeWriter, TPipeReader

Рисунок 4.22 Успішне проходження тестування TPipeChannel, TPipeWriter, TPipeReader

Проведемо тестування за таблицею 4.2 TDeque. На рисунку 4.23 а 4.24 показано результати проведення та код тесту.

Таблиця 4.2 Тестування TDeque

Тест	Перевірка роботи TDeque
Номер тесту	2
Початковий стан	Маємо TDeque заповнений декількома векторами
Вхідні дані	Пари початкових та кінцевих індексів: {0, 15}, {3, 12}
Опис проведення тесту	Викликаємо метод Loop, передаємо початкові та кінцеві ідекси.
Очікуваний результат	Перевіряємо, чи справді початкові та кінцеві індекси в кінці будуть рівні.
Фактичний результат	Вони рівні.

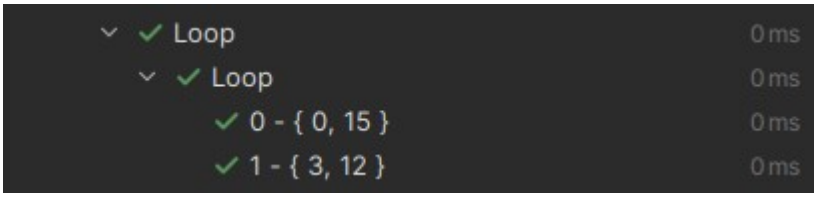

Рисунок 4.23 Успішне проходження тестування TDeque

```
class TDequeTest
    : public testing::TestWithParam<std::array<int, 2>> {
    protected:
    static void SetUpTestSuite();
    static bfs::TDeque<int> s_vDeque;
};

void TDequeTest::SetUpTestSuite() {
    s_vDeque.Push({0, 1, 2, 3, 4});
    s_vDeque.Push({5, 6, 7, 8, 9});
    s_vDeque.Push({10, 11, 12, 13, 14});
}

bfs::TDeque<int> TDequeTest::s_vDeque = bfs::TDeque<int>();

INSTANTIATE_TEST_SUITE_P(Loop, TDequeTest,
    testing::Values(std::array{0, 15}, std::array{3, 12}));

TEST_P(TDequeTest, Loop) {
    auto [beginIt, endIt] = GetParam();
    s_vDeque.Loop(beginIt, endIt, [&beginIt](const auto& el) {
        EXPECT_EQ(el, beginIt);
        ++beginIt;
    });
    EXPECT_EQ(beginIt, endIt);
}
```

Рисунок 4.24 Тест працездатності TDeque

Аналогічно до пункту 2.3 проводемо тестування алгоритмів TSharedBFS та TCommunicationBFS на коректність утвореного шляху за таблицею 4.3. Порівняння швидкості між ними та послідовним алгоритмом проведемо у наступному розділі. Наразі звернемо увагу лише на коректність.

Таблиця 4.3 Тестування алгоритмів TSharedBFS та TCommunicationBFS

Тест	Перевірка роботи TSharedBFS, TCommunicationBFS
Номер тесту	3
Початковий стан	Маємо початкові дані
Вхідні дані	Кількість потоків: 2, 3, 4, 5, 6, 7, 8, 9

	Розмір графів: 200, 300
Опис проведення тесту	У циклі для кожної кількості потоку, для кожного розміру графа пеперевіряємо чи повертається коректний шлях.
Очікуваний результат	Усі шляхи існують.
Фактичний результат	Усі шляхи існують.

```
for(const auto threadsNum : threadsNums) {
    const auto start = std::chrono::system_clock::now();
    const auto result = bfs::TSharedBFS<unsigned>::Do(grid, 0, lastIndex, threadsNum);
    const auto delay = std::chrono::system_clock::now() - start;
    const auto millis = std::chrono::duration_cast<std::chrono::milliseconds>(delay).count();
    EXPECT_TRUE(IsPathValid(result.value(), grid));
    WriteToReport(std::format("{{ \"name\": \"{}\", \"size\": {}, \"threadsNum\": {}, \"milliseconds\": {}, \"acceleration\": {} }},
        "Shared", size, threadsNum, millis, sequentialMillis / static_cast<double>(millis)));
}
for(const auto threadsNum : threadsNums) {
    const auto start = std::chrono::system_clock::now();
    const auto result = bfs::TCommunicationBFS<unsigned>::Do(grid, 0, lastIndex, threadsNum);
    const auto delay = std::chrono::system_clock::now() - start;
    const auto millis = std::chrono::duration_cast<std::chrono::milliseconds>(delay).count();
    EXPECT_TRUE(IsPathValid(result.value(), grid));
    WriteToReport(std::format("{{ \"name\": \"{}\", \"size\": {}, \"threadsNum\": {}, \"milliseconds\": {}, \"acceleration\": {} }},
        "Communication", size, threadsNum, millis, sequentialMillis / static_cast<double>(millis)));
}
```

Рисунок 4.25 Тестування коректності TSharedBFS та TCommunicationBFS

5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМІВ

Для усередненої оцінки ефективності алгоритмів для кожної комбінації параметрів усі тести було провдено 5 разів. Усі тести мали однаковий початок у лівому верхньому куті графа та кінець у правому нижньому куті графа. Тести проводилися на графах достатньо великого розміру, щоб можна було помітити суттєву різницю.

У таблиці 5.1 наведено результати тестування ефективності паралельних алгоритмів відносно послідовного.

Розмір	Послідовний, мс	Кількість потоків	Час виконання, мс		Прискорення	
			Shared	Communication	Shared	Communication
2500	5967.6	2	5565.8	4271.4	1.072495	1.397238
		3	4496.0	3452.8	1.327318	1.728319
		4	4192.0	3078.0	1.423594	1.939011
		5	4462.4	2915.6	1.337327	2.048491
		6	9079.4	2837.8	0.658027	2.104320
		7	8980.6	3059.8	0.664930	1.951420
		8	8936.8	2964.0	0.667879	2.013717
		9	8442.4	2896.0	0.707279	2.061587
2625	6643.2	2	6421.2	4726.0	1.034556	1.405724
		3	4985.8	3775.0	1.332415	1.759888
		4	4649.8	3402.8	1.428910	1.952944
		5	4915.2	3267.2	1.351557	2.033648
		6	10102.2	3170.4	0.657617	2.102071
		7	10001.6	3408.6	0.664219	1.949354
		8	9631.4	3340.0	0.693449	1.989316
		9	9281.0	3299.6	0.716052	2.014581
2750	7336.6	2	6982.6	5169.8	1.053011	1.419095
		3	5516.4	4184.2	1.330618	1.753511
		4	5096.6	3784.4	1.439843	1.939009
		5	5388.8	3622.2	1.361512	2.027397
		6	11044.8	3449.8	0.664284	2.127009
		7	10714.8	3736.6	0.685283	1.963517
		8	10960.4	3664.4	0.669394	2.002786
		9	10192.0	3557.2	0.720460	2.062910

2875	9370.8	2	8356.0	5688.0	1.100884	1.645216
		3	6011.6	4627.6	1.558826	2.026361
		4	5544.4	4039.2	1.690241	2.329854
		5	5875.4	3870.6	1.594627	2.421424
		6	11795.2	3698.6	0.793217	2.536562
		7	11869.2	3996.8	0.790064	2.339179
		8	11980.0	3896.8	0.781524	2.402154
		9	11044.8	4244.2	0.851188	2.293266
3000	8698.6	2	8075.6	6244.8	1.077135	1.393493
		3	6611.8	5025.6	1.316568	1.731800
		4	6051.8	4459.0	1.437368	1.953030
		5	6412.0	4190.6	1.356640	2.077352
		6	12989.6	4194.6	0.669717	2.080761
		7	12824.2	4433.2	0.678593	1.962312
		8	12975.0	4313.0	0.670554	2.016832
		9	12152.0	4192.8	0.715950	2.074758
3125	9284.8	2	9191.2	6753.0	1.010178	1.374907
		3	7144.6	5397.4	1.299556	1.720268
		4	6595.4	4789.2	1.407775	1.938705
		5	6929.6	4502.0	1.339878	2.062906
		6	14050.6	4414.4	0.660815	2.103987
		7	14127.4	4796.4	0.657246	1.935901
		8	14105.8	4659.4	0.658293	1.992710
		9	13323.0	4560.2	0.697122	2.036032
3250	10215.0	2	9542.8	7250.2	1.070470	1.408970
		3	7719.2	5841.4	1.323370	1.749109
		4	7092.0	5136.4	1.440383	1.988923
		5	7503.2	4885.0	1.361426	2.091544
		6	15123.2	4711.6	0.675465	2.168662
		7	15515.4	5178.4	0.658399	1.972640
		8	15382.8	5008.8	0.664083	2.039499
		9	14334.6	4904.6	0.712742	2.082795
3500	12120.2	2	11186.0	8505.2	1.083534	1.425279
		3	8956.0	6723.8	1.353326	1.803116
		4	8251.8	5919.0	1.468817	2.047721
		5	8624.6	5585.6	1.405320	2.169932
		6	17420.6	5403.6	0.695748	2.244636

		7	17752.2	5987.4	0.682755	2.024328
		8	17716.0	5780.4	0.684252	2.096769
		9	16580.4	5642.8	0.731146	2.147984
3635	13128.2	2	12664.6	9213.0	1.036596	1.425516
		3	9722.2	7319.4	1.350321	1.793807
		4	8956.6	6404.0	1.465800	2.050045
		5	9282.6	5973.6	1.414275	2.197908
		6	18555.8	5945.6	0.707512	2.208844
		7	19106.0	7028.4	0.687173	1.902146
		8	19061.0	7275.0	0.688758	1.908599
		9	18077.6	6948.6	0.726515	1.978160
3750	14052.8	2	13000.6	9944.0	1.080917	1.413471
		3	10418.2	7752.4	1.348877	1.813032
		4	9514.6	6843.0	1.477003	2.053873
		5	9927.0	6427.8	1.415651	2.186625
		6	19253.6	6377.0	0.731297	2.204796
		7	20283.0	6933.2	0.692959	2.026949
		8	20431.4	6673.4	0.687862	2.105961
		9	19045.2	6503.0	0.738258	2.161034
3875	15112.6	2	14489.8	10541.0	1.042998	1.434301
		3	11139.0	8374.6	1.356780	1.804565
		4	10214.6	7296.2	1.479506	2.071501
		5	10565.4	6896.4	1.430392	2.191925
		6	20722.8	6743.2	0.729397	2.242101
		7	21726.4	7370.6	0.695683	2.050412
		8	21804.8	7112.4	0.693162	2.124890
		9	20423.2	6919.4	0.740072	2.184081
4000	16670.8	2	15044.4	11915.6	1.108142	1.399129
		3	12021.6	9232.4	1.386714	1.805868
		4	10915.2	7909.0	1.527283	2.107924
		5	11257.8	7398.4	1.480799	2.253684
		6	21858.2	7291.0	0.762673	2.287835
		7	22849.6	7945.2	0.730115	2.098303
		8	23075.6	7624.0	0.722618	2.186598
		9	22141.0	7413.6	0.753037	2.248660

На рисунках 5.1, 5.2 показано залежність часу від розміру графа, прискорення від розміру графа при різній кількості потоків.

З вигляду графіків відразу можна побачити, що BFS з повідомленнями показує себе відмінно і дає прискорення трохи більше за два рази зі збільшенням кількості потоків.

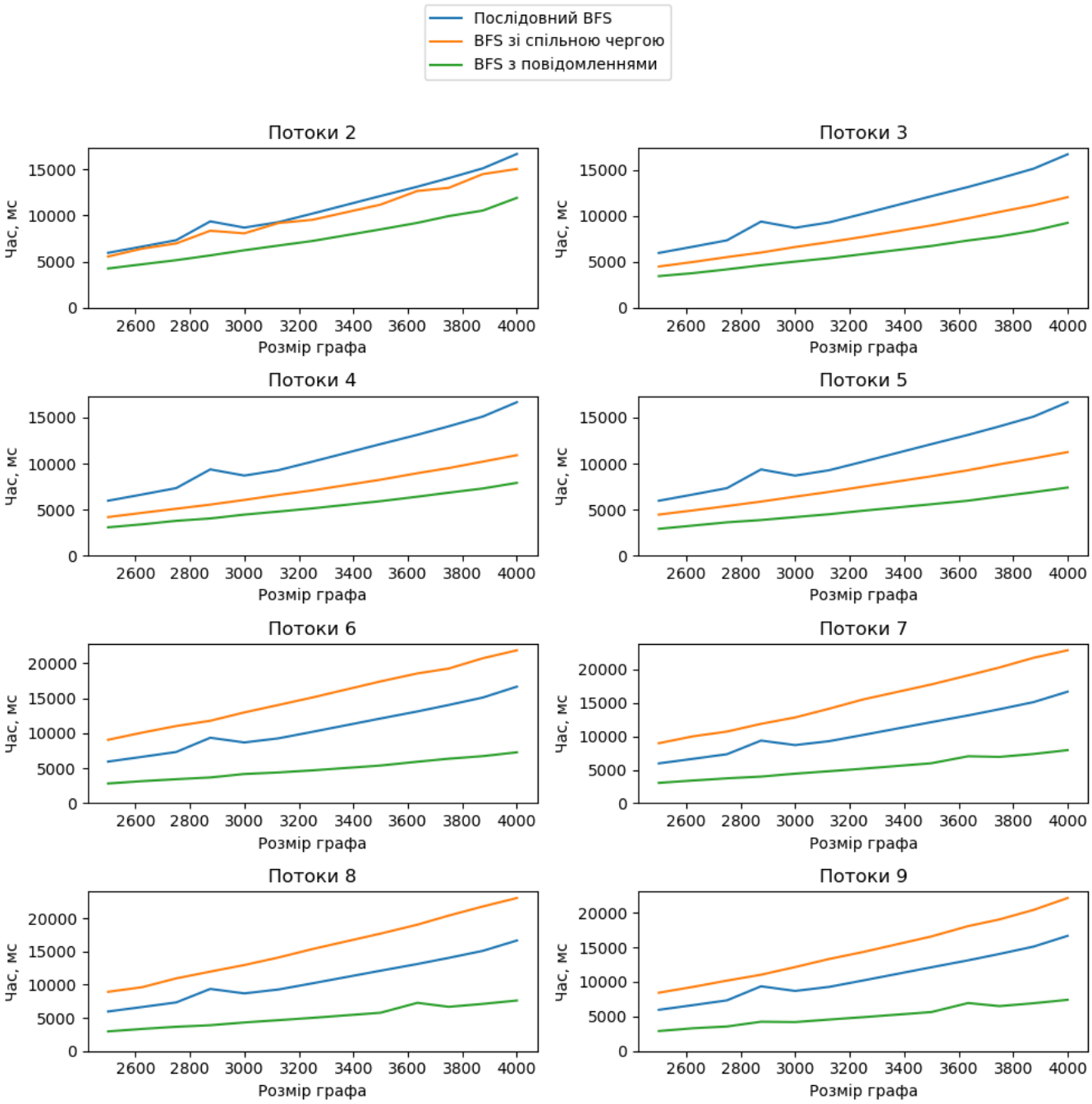

Рисунок 5.1 Залежність часу від розміру графа для різної кількості потоків

На відміну від BFS з повідомленнями BFS зі спільною чергою показує прискорення вище за 1.2, але при кількості потоків більше за 5, результати погані навіть гірші від послідовної версії на 25-30%. Даний факт можна пояснити тим, що

через спільну чергу потоки надто часто синхронізуються між собою, і вони заважають один одному.

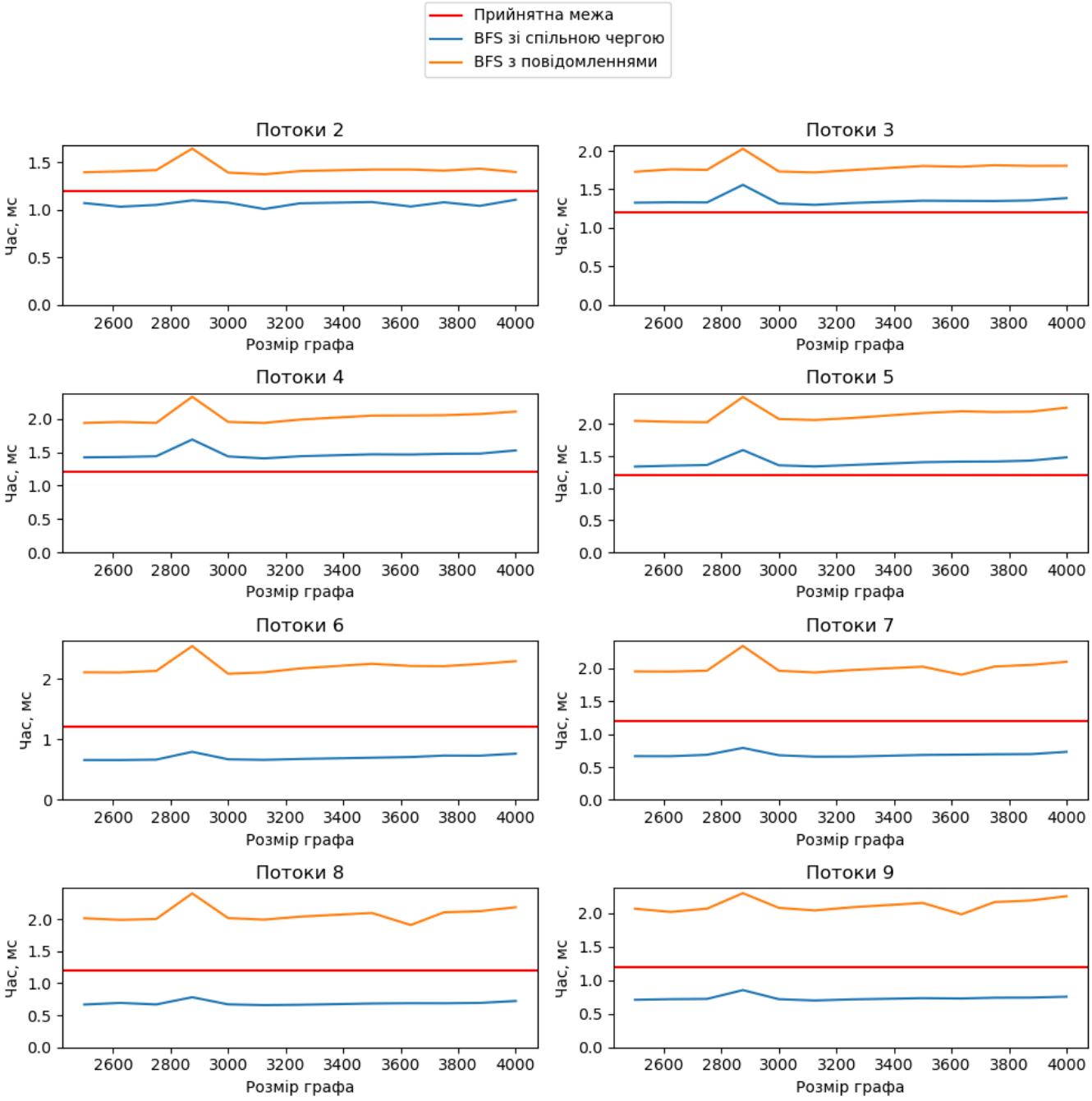

Рисунок 5.2 Залежність прискорення від розміру графа для різної кількості потоків

ВИСНОВКИ

Під час виконання курсової роботи було досліджено варіанти паралельної реалізації BFS. Були описані основні класи, допоміжні структури даних, демонстровано схеми їхньої роботи. Також проведене успішне їхнє тестування, що допомогло відловити некоректну роботу і переконатися, що все працює правильно.

Цікаво було розглянути власну реалізацію потокобезпечної черги та каналів, яких, на жаль, нема у стандартній бібліотеці C++.

Наведено також результати вимірювань часу та прискорення алгоритмів у таблицях та графіках, які можна перегрянути в розділі 5.

Загалом реалізація паралельних версій виявилася успішною, і їхнє прискорення більше за 1.2, однак для BFS зі спільною чергою ця умова не виконується для кількості потоків більшої за 5, що пояснюється надмірною синхронізацією між ними.

Як побачили на прикладі BFS зі спільною чергою, роботи певний ресурс спільним до модифікації між всіма потоками не є найкращою ідею і може суттєво вплинути на час роботи програми. З іншого боку, ми переконалися на прикладі BFS з повідомленнями, що розділення суцільної роботи на малі частини між дочірніми потоками, знижує витрати на синхронізацію і дає відчутне прискорення.